МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра вычислительной техники



**ОТЧЁТ**

**по лабораторной работе №3**

**по дисциплине:** *Алгоритмы и структуры данных*

*Тема:* **«***Коллекции данных – сбалансированное дерево поиска***»**

Выполнили:Проверил:

Студент гр. АВТ-710 *Ассистент каф. ВТ*

*Покалюк Е.А. Михайленко Д.А.*

«\_\_\_» \_\_\_\_\_\_ 2019г.«\_\_\_» \_\_\_\_\_\_ 2019 г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск 2020 г.

# 1. Цель работы

Изучение и исследование методов балансировки двоичных деревьев поиска на примере АТД «Сбалансированное дерево поиска». Освоение методики модификации коллекций с помощью механизма наследования классов.

# 2. Общее задание

1. Спроектировать, реализовать и провести тестовые испытания АТД «Сбалансированное дерево поиска» для коллекции, содержащей данные произвольного типа. Тип данных задаётся клиентской программой.

АТД «Сбалансированное дерево поиска» представляет собой модифицированную версию АТД «**BST**-дерево» с трудоёмкостью операций ***O(log*2*n).***

Интерфейс АТД «Сбалансированное дерево поиска» включает следующие операции:

* опрос размера дерева,
* очистка дерева,
* проверка дерева на пустоту,
* поиск объекта с заданным ключом,
* включение нового объекта с заданным ключом,
* удаление объекта с заданным ключом,
* запрос итератора произвольного доступа, установленного на узел дерева с минимальным ключом *begin(),*
* запрос итератора произвольного доступа*,* установленного на узел дерева с максимальным ключом *rbegin()*,
* запрос «неустановленного» итератора *end(),*

Операции итератора произвольного доступа для симметричного обхода элементов дерева:

* доступ по чтению и записи к данным текущего узла \*,
* переход к следующему узлу в дереве ++,
* переход к предыдущему узлу в дереве --,
* проверка равенства итераторов ==
* проверка неравенства итераторов !=

Для тестирования коллекции интерфейс АТД «Сбалансированное дерево поиска» включает дополнительные операции:

* вывод структуры дерева на экран,
* опрос числа просмотренных операцией узлов дерева.

1. Выполнить отладку и тестирование всех операций АТД «Сбалансированное дерево поиска» с помощью меню операций.
2. Выполнить сравнительное тестирование трудоёмкости операций вставки, удаления и поиска для коллекций «**BST**-дерево» и «Сбалансированное дерево поиска» для случайной и вырожденной структуры дерева.
3. Выполнить сравнительный анализ теоретических и экспериментальных показателей трудоёмкости операций.

**Вариант задания:** RB-дерево, как модификация BST-дерева. Алгоритмы операций вставки и поиска реализуются в рекурсивной форме, алгоритм удаления реализуется в итеративной форме.

# 3. Формат АТД: Красно-черное дерево поиска

Абстрактный тип данных указанного типа T, являющийся ассоциативной структурой, доступ к данным в которой происходит по значению ключа, который является уникальным идентификатором и может встречаться в коллекции лишь однократно.

ДАННЫЕ:

Параметры:

Текущий размер дерева **size**

Структура данных: бинарное дерево на основе адресных указателей.

**ОПЕРАЦИИ:**

**Конструктор:**

*Вход:* нет

*Процесс:* создание пустого RB–дерева

*Начальные значения:*size = 0

*Постусловия***:** создано новое дерево, size = 0

**Конструктор копирования:**

*Вход:* ссылка на копируемое дерево

*Предусловия:* нет

*Процесс:*копирование дерева

*Выход:* нет

*Постусловия:* создана копия дерева

**Опрос размера дерева:**

*Вход:* нет

*Предусловия:* нет

*Процесс:* чтение *size*

*Выход: size*

*Постусловия:* нет

**Очистка дерева:**

*Вход:* нет

*Предусловия:* нет

*Процесс:* удаления узлов дерева

*Выход:* нет

*Постусловия:* дерево пусто, *size* = 0

**Проверка дерева на пустоту:**

*Вход:* нет

*Предусловия:* нет

*Процесс:* чтение текущего размера дерева

*Выход:* true, если дерево пустое – *size* = 0

false, в противном случае – *size* ≠ 0

*Постусловия:* нет

**Доступ к данным по ключу:**

*Вход:* ключ искомого элемента *key*

*Предусловия:* существование узла в заданным key

*Процесс:* поиск данных по ключу *key*

*Выход:* данные с ключом *key*; генерация исключения при невыполнении предусловия

*Постусловия:* нет

**Включение данных с заданным ключом:**

*Вход:* ключ добавляемого элемента *key* и его данные *data*

*Предусловия:* заданный *key* отсутствует в существующем дереве

*Процесс:* добавление элемента с ключом *key* и данными *data*

*Выход:* true – элемент включён успешно,

false **–** при невыполнении предусловия

*Постусловия:* *data* добавлен в дерево по ключу *key, size++*

**Удаление данных с заданным ключом:**

*Вход:* ключ удаляемого элемента *key*

*Предусловия:* заданный *key* присутствует в существующем дереве

*Процесс:* удаление элемента с ключом *key*

*Выход:* true – элемент удалён успешно,

false **–** при невыполнении предусловия

*Постусловия:* узел с данным key удалён из дерева, *size--*

**Операция создания итератора произвольного доступа begin( )**

*Вход*: нет

*Предусловия*: размер существующего дерева *size* ≠ 0

*Процесс*: создание итератора произвольного доступа, установленного на первый элемент последовательности в дереве

*Выход*: итератор произвольного доступа установленный на первый элемент begin( ) или «неустановленный» итератор end( ) при невыполнении предусловия

*Постусловия*: нет

**Запрос «неустановленного» итератора end( )**

*Вход*: нет

*Предусловия*: нет

*Процесс*: чтение «не установленного» итератора, указывающего на позицию, следующую за последним элементом последовательности в дереве

*Выход*: «не установленный» итератор произвольного доступа end( )

*Постусловия*: нет

**Операция создания итератора произвольного доступа rbegin( )**

*Вход*: нет

*Предусловия*: размер существующего дерева *size* ≠ 0

*Процесс*: создание итератора произвольного доступа, установленного на последний элемент

*Выход*: итератор произвольного доступа к элементам rbegin( ) или «неустановленный»

*Постусловия*: нет

# 4. Формат АТД: Итератор BST-Дерева

Итератор – объект, используемый для доступа к значениям узлов дерева и для перемещения по дереву. В данной работе при создании итератор сразу должен привязываться к дереву. Итератор реализует доступ к узлам дерева порядке возрастания ключей. Так как RBTree является наследником Tree, этот итератор может использоваться и для обхода красно-чёрного дерева.

Имеется возможность установки итератора на первый узел дерева, последний узел дерева, перемещение к следующему или предыдущему элементу дерева.

Существует 2 состояния итератора: установлен (в пределах дерева) и не установлен (за пределами дерева).

**ДАННЫЕ:**

*Параметры:*

Указатель на дерево *BST<Key, Data>\* tree*

Указатель на узел *Node<Key, Data>\* node*

**ОПЕРАЦИИ:**

**Установка итератора на следующий узел дерева:**

*Вход:* нет

*Предусловия:* итератор находится в состоянии «установлен»

*Процесс:*установка итератора на следующий узел дерева

*Выход:* установление итератора на следующий элемент, генерация исключения при невыполнении предусловия

*Постусловия:* итератор установлен на следующий узел дерева

**Установка итератора на предыдущий узел дерева:**

*Вход:* нет

*Предусловия:* итератор находится в состоянии «установлен»

*Процесс:*установка итератора на предыдущий узел дерева

*Выход:* установка итератора на предыдущий узел дерева, генерация исключения при невыполнении предусловия

*Постусловия:* итератор установлен на предыдущий узел дерева

**Доступ по чтению и записи к текущему значению:**

*Вход*: нет

*Предусловие*: итератор в состоянии «установлен»

*Процесс*: чтение/запись нового значения на место текущего

*Выход*: ссылка на данные элемента или nullptr – исключение

*Постусловие*: нет

**Проверка равенства итераторов:**

*Вход*: нет

*Предусловие*: нет

*Процесс*: проверка на равенство итераторов

*Выход*: true – равны, false – не равны

*Постусловие*: нет

**Проверка неравенства итераторов:**

*Вход*: нет

*Предусловие*: список не пуст

*Процесс*: проверка на неравенство итераторов

*Выход*: true – не равны, false – равны

*Постусловие*: нет

# 5. Шаблонный класс для коллекции «Сбалансированное дерево поиска»

template <typename Key, typename Data>

class RBTree : virtual public Tree <Key, Data>

{

public:

RBTree() : Tree <Key, Data>::Tree() {} // конструктор

RBTree(const Tree<Key, Data> &tree) : Tree <Key, Data>::Tree(tree){} // конструктор копирования

virtual ~RBTree(); // деструктор

bool insert(Data obj, Key key); // вставка нового узла с заданными ключом и значением

bool deleteByKey(Key key); // удаление узла с заданным ключом

private:

bool isRed(typename Tree <Key, Data>::Node\* node); // проверка на цвет узла

void setColor(typename Tree <Key, Data>::Node\* node, bool color); // установка цвета узла

typename Tree <Key, Data>::Node\* getLeft(typename Tree <Key, Data>::Node\* node); // вернуть левого потомка

typename Tree <Key, Data>::Node\* getRight(typename Tree <Key, Data>::Node\* node); // вернуть правого потомка

typename Tree <Key, Data>::Node\* getParent(typename Tree <Key, Data>::Node\* node); // вернуть родителя

protected: // вспомогательные функции

typename Tree <Key, Data>::Node\* rotate(typename Tree <Key, Data>::Node\* p, typename Tree <Key, Data>::Side side); // поворот

bool insert(typename Tree <Key, Data>::Node\*& node, Key key, Data value, typename Tree <Key, Data>::Side side); // вставка

void fixRBTree(typename Tree <Key, Data>::Node\* x); // сохранение свойств красно-черного дерева

# };

# 6. Тестирование трудоёмкости операций

Тестирование трудоёмкости дерева проводилось для двух случаев: вырожденного и случайно заполненного дерева

1. Создаем генератор случайных чисел большой разрядности, который будем использовать для заполнения массива и дерева соответственно элементами типа long long со случайными ключами
2. Создаем тест трудоемкости операций случайных BST и RB – деревьев, где

BST< INT\_64,int > tree – дерево для 64 – разрядных ключей типа INT\_64,

INT\_64\* m=new INT\_64 [n] – массив для ключей, которые присутствуют в дереве,

* Функция sRand – установка первого случайного числа от часов компьютера,
* Заполняем деревья и массив элементами со случайными ключами,
* Выводим размер деревьев до теста,
* Обнуляем счетчики трудоемкости для операций,
* Генерируем поток операций, 10% - промахи операций, 90% - успешных операций,
* Обрабатываем исключение при ошибка операции поиска,
* Выводим размер деревьев после теста, теоретическую оценку, экспериментальную оценку вставки, удаления, поиска,
* Освобождаем память массива.
* Повторяем для размеров деревьев: 100, 500, 1000, 2000, 5000, 10000.

1. Для вырожденных BST и RB-деревьев заполнение массива и дерева элементами с возрастающими четными ключами на интервале [0, 10000, 20000, …, 10000\*n].
2. Полученные значения заносим в таблицы 1, 2 и 3, 4. По значениям в таблице строим графики зависимости трудоемкости вставки, удаления и поиска от количества элементов в дереве.

Таблица 1. Трудоемкость для вырожденого BST дерева

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | 1.39 log2(n) | Вставка | Удаление | Поиск |
| 1000 | 13,8524 | 13,532 | 12,176 | 11,496 |
| 3000 | 16,0555 | 16,694 | 15,266 | 14,9027 |
| 5000 | 17,079 | 17,193 | 15.89 | 15.414 |
| 7000 | 17,7547 | 17,5857 | 16,306 | 15,6551 |
| 10000 | 18,4699 | 18.0298 | 16,865 | 16,1818 |
| 15000 | 19,283 | 19,2483 | 18,1697 | 17,4767 |

Таблица 2. Трудоемкость для вырожденого RB дерева

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | 1.02 log2(n) | Вставка | Удаление | Поиск |
| 1000 | 10,1651 | 10,268 | 9,756 | 10,106 |
| 3000 | 11,7818 | 11,874 | 11,466 | 12,049 |
| 5000 | 12,5335 | 12,619 | 12,247 | 12,87 |
| 7000 | 13,0286 | 13,102 | 12,7586 | 13,3594 |
| 10000 | 13,5535 | 13,6436 | 13,3052 | 13,9234 |
| 15000 | 14,1501 | 14,216 | 13,9257 | 14,5671 |

*Рисунок 1.* *Сравнение трудоемкостей операций вырожденного дерева*

Таблица 3. Трудоемкость для случайного BST дерева

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | BST | Вставка | Удаление | Поиск |
| 50 | 25 | 18,68 | 17,64 | 15,24 |
| 100 | 50 | 38,9 | 45,36 | 45,38 |
| 150 | 75 | 64,1867 | 65,1876 | 67,16 |
| 200 | 100 | 76,83 | 80,06 | 81,91 |

Таблица 4. Трудоемкость для случайного RB дерева

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | RB | Вставка RB | Удаление RB | Поиск RB |
| 50 | 5,7567 | 5,2 | 5,6 | 5,4 |
| 100 | 6,7767 | 6,3 | 7,28 | 6,8 |
| 150 | 7,3734 | 7,066 | 7,813 | 7,146 |
| 200 | 7,7967 | 7,61 | 8,12 | 8 |

*Рисунок 2.* *Сравнение трудоемкостей операций случайного дерева*

# Выводы

В результате выполнения лабораторной работы был создан абстрактный тип данных — красно-черное дерево поиска. Работоспособность класса была проверена с помощью созданной программы-меню, дающей доступ ко всем методам созданного класса.

Созданный класс был протестирован на производительность путём нахождения трудоёмкости вставки, удаления и поиска элементов в коллекцию для случайно сгенерированного и вырожденного деревьев со случайно генерируемыми ключами.

Тестирование продемонстрировало следующие результаты: для вставки, удаления и поиска трудоемкость равна 1.002\*log2N, если дерево генерируется случайным образом, похожая ситуация наблюдается при вырожденном случае т.к дерево сбалансированное.

# Список литературы

1) Проектирование и реализация коллекций данных: учебно-методическое пособие / Т.А. Романенко ; Новосиб. гос.техн.ун-т

2) Абстракция данных и решение задач на С++. Стены и зеркала / Фрэнк М. Каррано, Джанет Дж. Причардт; [пер. С англ. И ред. Д.А. Клюшина]

# Приложение

**RBTree.h**

#pragma once

#include <iostream>

#include <stack>

#include "Tree.h"

using namespace std;

template <typename Key, typename Data>

class RBTree : virtual public Tree <Key, Data>

{

public:

RBTree() : Tree <Key, Data>::Tree() {} // конструктор

RBTree(const Tree<Key, Data> &tree) : Tree <Key, Data>::Tree(tree){} // конструктор копирования

virtual ~RBTree(); // деструктор

bool insert(Data obj, Key key); // вставка нового узла с заданными ключом и значением

bool deleteByKey(Key key); // удаление узла с заданным ключом

private:

bool isRed(typename Tree <Key, Data>::Node\* node); // проверка на цвет узла

void setColor(typename Tree <Key, Data>::Node\* node, bool color); // установка цвета узла

typename Tree <Key, Data>::Node\* getLeft(typename Tree <Key, Data>::Node\* node); // вернуть левого потомка

typename Tree <Key, Data>::Node\* getRight(typename Tree <Key, Data>::Node\* node); // вернуть правого потомка

typename Tree <Key, Data>::Node\* getParent(typename Tree <Key, Data>::Node\* node); // вернуть родителя

protected: // вспомогательные функции

typename Tree <Key, Data>::Node\* rotate(typename Tree <Key, Data>::Node\* p, typename Tree <Key, Data>::Side side); // поворот

bool insert(typename Tree <Key, Data>::Node\*& node, Key key, Data value, typename Tree <Key, Data>::Side side); // вставка

void fixRBTree(typename Tree <Key, Data>::Node\* x); // сохранение свойств красно-черного дерева

};

template <typename Key, typename Data>

bool RBTree<Key, Data>::isRed(typename Tree <Key, Data>::Node\* node)

{

return node ? (node->color == RED) : false;

}

template <typename Key, typename Data>

typename Tree <Key, Data>::Node\* RBTree<Key, Data>::getLeft(typename Tree <Key, Data>::Node\* node)

{

return node ? node->left : NULL;

}

template <typename Key, typename Data>

typename Tree <Key, Data>::Node\* RBTree<Key, Data>::getRight(typename Tree <Key, Data>::Node\* node)

{

return node ? node->right : NULL;

}

template <typename Key, typename Data>

typename Tree <Key, Data>::Node\* RBTree<Key, Data>::getParent(typename Tree <Key, Data>::Node\* node)

{

return node ? node->parent : NULL;

}

template <typename Key, typename Data>

void RBTree<Key, Data>::setColor(typename Tree <Key, Data>::Node\* node, bool color) {

if (node) node->color = color;

}

template <typename Key, typename Data>

RBTree<Key, Data>::~RBTree()

{

this->clear(this->root);

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* RBTree<Key, Data>::rotate(typename Tree <Key, Data>::Node\* p, typename Tree <Key, Data>::Side side)

{

if (!p) return NULL;

if (side == this->LEFT) // в случае поворота налево

{

typename Tree <Key, Data>::Node\* r = p->right; // Node\* r - стержень дерева

p->right = r->left;

if (r->left) r->left->parent = p;

r->parent = p->parent; // при этом возможно что r станет корнем дерева,

if (!p->parent) // если нет предка у узла p

{

this->root = r;

}

else if (p->parent->left == p) // если левый потомок предка p ссылается сам на себя,

{

p->parent->left = r; // то присваиваем ему значение стержня

}

else

{

p->parent->right = r;

}

r->left = p; // делаем p левым потомком стержня r

p->parent = r; //

return r;

}

else // в случае поворота направо аналогично

{

typename Tree <Key, Data>::Node\* l = p->left;

p->left = l->right;

if (l->right) l->right->parent = p;

l->parent = p->parent;

if (!p->parent)

{

this->root = l;

}

else if (p->parent->right == p)

{

p->parent->right = l;

}

else

{

p->parent->left = l;

}

l->right = p;

p->parent = l;

return l;

}

}

template <typename Key, typename Data>

bool RBTree<Key, Data>::insert(typename Tree <Key, Data>::Node\*& node, Key key, Data value, typename Tree <Key, Data>::Side side)

{

this->cnt++;

if (!node) // при отсутствии заданного узла

{

node = new typename Tree <Key, Data>::Node(value, key); // создаем его

this->cnt++;

return true;

}

if (isRed(node->left) && isRed(node->right)) // когда узел существует, а его левый и правый потомки красные

{

node->color = RED; // цвет узла будет красным

node->left->color = BLACK; // а цвета потомков черным (свойство RB-дерева)

node->right->color = BLACK; //

}

bool success; // успешно ли прошла вставка

if (node->key == key) // если такой ключ уже есть в данном узле возврат false

{

return false;

}

else if (key < node->key) // иначе если заданный ключ меньше ключа в узле

{

success = insert(node->left, key, value, this->LEFT); // рекурсивно вставляем влево

node->left->parent = node;

if (isRed(node) && isRed(node->left) && (side == this->RIGHT)) // если узел и его левый потомок красные,

{

node = rotate(node, this->RIGHT); // то поворачиваем вправо

}

if (isRed(node->left) && isRed(node->left->left)) // если левый потомок и левый потомок левого потомка красные

{

node = rotate(node, this->RIGHT); // поворачиваем вправо

node->color = BLACK; // красим узел в черный

node->right->color = RED; // а его правого потомка в красный

}

}

else

{

success = insert(node->right, key, value, this->RIGHT); // иначе рекурсивно вставляем вправо

node->right->parent = node;

if (isRed(node) && isRed(node->right) && (side == this->LEFT)) // если узел и его правый потомок красные, а сторона левая,

{

node = rotate(node, this->LEFT); // то поворачиваем влево

}

if (isRed(node->right) && isRed(node->right->right)) // если правый потомок и правый потомок правого потомка красные

{

node = rotate(node, this->LEFT); // поворачиваем влево

node->color = BLACK; // красим узел в черный

node->left->color = RED; // а левого потомка в красный

}

}

return success;

}

template <typename Key, typename Data>

bool RBTree<Key, Data>::insert(Data value, Key key)

{

this->cnt = 0;

bool success = insert(this->root, key, value, this->LEFT); // вставка по умолчанию влево

if (!success) return false;

this->root->color = BLACK; // красим корень в черный

this->size++; // инкремент счетчика размера дерева

return true;

}

template <typename Key, typename Data>

bool RBTree<Key, Data>::deleteByKey(Key key)

{

this->cnt = 0;

typename Tree <Key, Data>::Node \*t = this->root, \*x = NULL; // x - вспомогательная переменная

while (t != NULL && t->key != key) // пока не дошли до листа с NULL-значением и ключ корня не равен заданному

{

if (key < t->key) t = t->left; // если заданный ключ меньше текущего то идем влево

else t = t->right; // иначе вправо

this->cnt++;

}

if (t == NULL) return false; // если дошли до листа

this->cnt++;

if(t->left && t->right) // если есть правый и левый потомки

{

x= this->successor(t); // в x записываем следующий по ключу узел

t->key = x->key; // ключом t становится ключ x

t->data = t->data; // аналогично данные

t = x; // и сам узел

}

typename Tree <Key, Data>::Node\* replacement = (t->left ? t->left : t->right); // если есть левый потомок - записываем его, иначе - правого

if (replacement)

{

t->key = replacement->key; // запись в t ключей replacement

t->data = replacement->data; // данных

t->left = t->right = NULL; // обнуление потомков t

t = replacement;

}

else if (!t->parent) // если у t нет предка

{

this->root = NULL; // обнуляем корень

}

else

{

if (t->color == BLACK) // если t - черный, восстанавливаем RB-дерево согласно свойствам

fixRBTree(t);

if (t->parent) // если у t есть предок

{

if (t == t->parent->left) // если t - левый потомок предка t,

t->parent->left = NULL; // то обнуляем левого потомка предка t

else if (t == t->parent->right) // если правый

t->parent->right = NULL; // аналогично

t->parent = NULL; // обнуляем предка t

}

}

delete t;

this->size--;

return true;

}

template <typename Key, typename Data>

void RBTree<Key, Data>::fixRBTree(typename Tree <Key, Data>::Node\* x)

{

while (x != this->root && isRed(x) == BLACK) // пока заданный узел x - не корень и он черный

{

if (x == getLeft(getParent(x))) // если x - левый потомок предка самого себя

{

typename Tree <Key, Data>::Node\* sibling = getRight(getParent(x)); // то братом x делаем правого потомка предка x

if (isRed(sibling) == RED) { // если брат x красный

setColor(sibling, BLACK); // делаем брата черным

setColor(getParent(x), RED); // делаем предка x красным

rotate(getParent(x), this->LEFT); // и поворачиваем влево по предку x

sibling = getRight(getParent(x)); // братом становится правый потомок предка x

}

if (!isRed(getLeft(sibling)) && !isRed(getRight(sibling))) { // если оба потомка брата черные

setColor(sibling, RED); // делаем брата красным

x = getParent(x); // а x становится предком

} else {

if (isRed(getRight(sibling)) == BLACK) { // если правый потомок брата черный

setColor(getLeft(sibling), BLACK); // красим левого потомка брата тоже в черный

setColor(sibling, RED); // а брата делаем красным

rotate(sibling, this->RIGHT); // поворачиваем вправо по брату

sibling = getRight(getParent(x)); // брат становится правым потомком предка x

}

setColor(sibling, isRed(getParent(x))); // цвет брата становится цветом отца x

setColor(getParent(x), BLACK); // красим предка x в черный

setColor(getRight(sibling), BLACK); // и правого потока брата в черный

rotate(getParent(x), this->LEFT); // поворачиваем влево по брату

x = this->root; // x становится корнем

}

} else { // симметрично

typename Tree <Key, Data>::Node\* sibling = getLeft(getParent(x));

if (isRed(sibling) == RED) {

setColor(sibling, BLACK);

setColor(getParent(x), RED);

rotate(getParent(x), this->RIGHT);

sibling = getLeft(getParent(x));

}

if (isRed(getRight(sibling)) == BLACK &&

isRed(getLeft(sibling)) == BLACK) {

setColor(sibling, RED);

x = getParent(x);

} else {

if (isRed(getLeft(sibling)) == BLACK) {

setColor(getRight(sibling), BLACK);

setColor(sibling, RED);

rotate(sibling, this->LEFT);

sibling = getLeft(getParent(x));

}

setColor(sibling, isRed(getParent(x)));

setColor(getParent(x), BLACK);

setColor(getLeft(sibling), BLACK);

rotate(getParent(x), this->RIGHT);

x = this->root;

}

}

}

setColor(x, BLACK);

}

**Tree.h**

#pragma once

#include <iostream>

#include <stack>

#include <iomanip>

using namespace std;

#define BLACK false

#define RED true

template <typename Key, typename Data>

class Tree

{

protected:

enum Side { LEFT, RIGHT }; // сторона узла

extern class Node;

Node \*root; // указатель на корень

int size; // размер дерева

int cnt; // количество пройденных узлов (для тестирования)

class Node

{

public:

Key key; // ключ узла

Data data; // значение в узле

Node \*left; // указатель на левого потомка

Node \*right; //указатель на правого потомка

Node \*parent;

int n; // количество узлов в поддереве

bool color;

Node(Data d, Key k) // конструкторы

{

key = k;

data = d;

left = right = parent = NULL;

n = 1;

color = RED;

}

Node()

{

key = NULL;

data = NULL;

left = right = parent = NULL;

n = 1;

color = RED;

}

};

public:

Tree(); // конструктор

Tree(const Tree<Key, Data>& tree); // конструктор копирования

virtual ~Tree(); // виртуальный деструктор

int getSize(); // опрос размера дерева

void clear(Node \*r); // очистка дерева

bool isEmpty(); // проверка дерева на пустоту

Data getData(Key key); // поиск значения узла с заданным ключом

bool setData(Data obj, Key key); // включение нового узла с заданными значениями и ключом

bool remove(Key key); // удаление узла с заданным ключом

void traverse(); // обход

void printTree(); // форматированный вывод дерева на экран

Key search(int k); // поиск по ключу

int getCount(); // подсчет узлов

Node\* getRoot() { return root; } // получение указателя на корень

class Iterator

{

private:

Tree\* tree; // указатель на дерево

Node\* node; // указатель на узел дерева

public:

Iterator(); // конструктор без параметров

Iterator(Tree<Key, Data>\* t, Node\* n); // конструктор с параметрами

Iterator(const Iterator& iter); // конструктор копирования

Data& operator\*(); // перегрузка указателя (доступ для чтения/записи)

Iterator& operator++(int); // переход к следующему узлу

Iterator& operator--(int); // переход к предыдущему узлу

bool operator==(Iterator& iterator); // проверка равенства итераторов

bool operator!=(Iterator& iterator); // проверка неравенства итераторов

bool isInside(); // проверка нахождения итератора внутри дерева

};

friend class Iterator;

Iterator begin(); // установка итератора на первый узел дерева

Iterator rbegin(); // установка итератора на последний узел дерева

Iterator end(); // запрос "неустановленного" итератора

protected:

void showTreeHelper(Node \*r, int level); // вспомогательный метод для вывода дерева

void traverseHelper(Node \*r); // обход узлов Lt-t-Rt

void copyTree(Node\* r); // вспомогательная функция для копирования дерева

Node\* predecessor(Node \*t); // поиск предыдущего по ключу узла

Node\* max(Node \*x); // узел с максимальным ключом

Node\* rightParent(Node \*t, Node \*x);

Node\* successor(Node \*t); // поиск следующего по ключу узла

Node\* min(Node \*t); // поиск узла с минимальным ключом

Node\* leftParent(Node \*t, Node \*x);

Node\* child(Node\* r, Node\* node);

Node\* parent(Node\* r, Node\* x);

Node\* searchHelper(Node \*t, int k); // вспомогательный метод поиска

void nullRoot(); // обнуление корня, вспомогательная функция

};

template <typename Key, typename Data>

Tree<Key, Data>::Tree()

{

root=NULL;

size=0;

cnt = 0;

}

template <typename Key, typename Data>

Tree<Key, Data>::Tree(const Tree<Key, Data>&tree)

{

root = NULL;

size = 0;

copyTree(tree.root);

}

template <typename Key, typename Data>

Tree<Key, Data>::~Tree()

{

clear(root);

}

template<typename Key, typename Data>

void Tree<Key, Data>::nullRoot()

{

root = NULL;

}

template <typename Key, typename Data>

void Tree<Key, Data>::clear(Node\* r) // очистка дерева рекурсивная

{

if (r)

{

if (r->left)

clear(r->left);

if (r->right)

clear(r->right);

delete r;

r = NULL;

}

nullRoot();

size = 0;

}

template <typename Key, typename Data>

void Tree<Key, Data>::copyTree(typename Tree<Key, Data>::Node\* r)

{

if (r == NULL)

return;

setData(r->key, r->data);

copyTree(r->left);

copyTree(r->right);

}

template <typename Key, typename Data>

bool Tree<Key, Data>::setData(Data val, Key key)

{

if (root == NULL) // если корень пуст, вставляем в него

{

root = new Node(val, key);

size++;

return true;

}

Node \*t = root, \*pred = NULL;

while (t!= NULL) // пока не дойдем до конечного листа

{

pred = t;

if (key == t->key) // ищем нужный ключ

{

t = root;

while (t->key != key)

{

pred = t;

if (key < t->key) // если заданный ключ меньше текущего то движемся влево

{

t->n = t->n-1;

t = t->left;

}

else // иначе вправо

{

t->n = t->n-1;

t = t->right;

}

}

return false; // если ключи совпадают

}

cnt++;

if (key < t->key)

{

t->n = t->n+1;

t = t->left;

}

else

{

t->n = t->n+1;

t = t->right;

}

}

if (key < pred->key)

pred->left = new Node(val, key);

else

pred->right = new Node(val, key);

size++;

cnt++;

return true;

}

//template <typename Key, typename Data>

//void Tree<Data,Key>::Show(Node \*r, int level)//вспомогательная функция для вывода структуры

//{

// if(r == NULL)

// return;

// Show(r->right, level+1);

// for(int i = 0; i <= 3\*level; i++)

// cout << " ";

// cout << r->data;

// (r->color==BLACK)? cout<<"B":cout<<"R"<< endl;

// Show(r->left, level+1);

//}

template <typename Key, typename Data>

void Tree<Key, Data>::showTreeHelper(Node\* r, int level) // вспомогательная функция для вывода форматированной структуры

{

if (r == NULL)

return;

if (right)

showTreeHelper(r->right, level + 4);

cout << setw(level) << ' ';

cout << r->data;

(r->color == BLACK) ? cout << "B" : cout << "R" << endl;

cout << endl;

if (left)

showTreeHelper(r->left, level + 4);

}

template <typename Key, typename Data>

bool Tree<Key, Data>::remove(Key key)

{

cnt = 0;

Node \*t = root, \*pred = NULL, \*x = NULL, \*t0 = NULL;

while (t!= NULL && t->key != key) // пока не дойдем до нужного ключа и не обнулим узел

{

cnt++;

pred = t;

if (key < t->key)

{

t->n = t->n-1;

t = t->left;

}

else

{

t->n = t->n-1;

t = t->right;

}

}

cnt++;

if (t == NULL) // если ключи совпадают - откатиться назад

{

t = root;

while (t != NULL && t->key != key)

{

pred = t;

if (key < t->key)

{

t->n = t->n+1;

t = t->left;

}

else

{

t->n=t->n+1;

t = t->right;

}

}

return false;

}

if (t->left != NULL && t->right != NULL) // если левый и правый потомки не пустые

{

t0 = t;

pred = t;

t->n = t->n-1;

t = t->right;

while (t->left != NULL)

{

cnt++;

pred = t;

t->n=t->n-1;

t = t->left;

}

cnt++;

x = t->right;

}

else

{

t0 = NULL;

if (t->left == NULL && t->right == NULL) x = NULL; // Если узел - лист

else if (t->left == NULL)

{

if (t->left!=NULL) x->n = x->n-1; x = t->right;

}

else if (t->right == NULL)

{

if (t->right!=NULL) x->n = x->n-1; x = t->left;

}

}

if (pred == NULL) root = x;

else

{

if (t->key < pred->key) pred->left = x;

else pred->right = x;

}

if (t0 != NULL)

{

t0->key = t->key;

t0->data = t->data;

}

delete t;

size--;

return true;

}

template <typename Key, typename Data>

void Tree<Key, Data>::printTree() // вывод структуры дерева на экран

{

if(root == NULL)

{

return;

}

showTreeHelper(root, 0);

}

template <typename Key, typename Data>

int Tree<Key, Data>::getSize()

{

return size;

}

template <typename Key, typename Data>

bool Tree<Key, Data>::isEmpty()

{

return (size == 0);

}

template <typename Key, typename Data>

Data Tree<Key, Data>::getData(Key key)

{

Node \*t = root;

while(t != NULL && t->key!=key)

{

if (key < t->key) t = t->left;

else t = t->right;

cnt++;

}

if (t == NULL) throw -1;

else return t->data;

}

template <typename Key, typename Data>

void Tree<Key, Data>::traverseHelper(Node \*root)

{

if(root==NULL) return;

traverseHelper(root->left);

cout << root->key << " ";

traverseHelper(root->right);

}

template <typename Key, typename Data>

void Tree<Key, Data>::traverse()

{

traverseHelper(root);

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::predecessor(typename Tree<Key, Data>::Node \*x)

{

if (x == NULL) return NULL;

if (x->left != NULL)

return max(x->left);

else return rightParent(root, x);

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::max(typename Tree<Key, Data>::Node \*t)

{

if (t==NULL) return NULL;

while (t->right!= NULL)

{

t=t->right;

}

return t;

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::rightParent(typename Tree<Key, Data>::Node \*t, typename Tree<Key, Data>::Node \*x)

{

if (t == x) return NULL;

if (x->key>t->key) {

Node \*rp = rightParent(t->right, x);

if (rp != NULL) return rp;

else return t;

}

else return rightParent(t->left, x);

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::successor(typename Tree<Key, Data>::Node \*x)

{

if (x == NULL) return NULL;

if (x->right != NULL) return min(x->right);

else return leftParent(root, x);

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::min(typename Tree<Key, Data>::Node \*t)

{

if (t == NULL) return NULL;

while (t->left != NULL) {t = t->left; cnt++;}

return t;

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::leftParent(typename Tree<Key, Data>::Node \*t, typename Tree<Key, Data>::Node \*x)

{

if (t == x) return NULL;

cnt++;

if (x->key<t->key)

{

cnt++;

Node \*lp = leftParent(t->left, x);

if (lp != NULL) return lp;

else return t;

}

else return leftParent(t->right, x);

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::child(typename Tree<Key, Data>::Node\* r, typename Tree<Key, Data>::Node\* node)

{

if (!node) return NULL;

else if (node->right) // node = node->right;

{

node = node->right;

while (node->left)

node = node->left;

return node;

}

else

{

node = leftParent(r, node);

return node;

}

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::parent(typename Tree<Key, Data>::Node\* r, typename Tree<Key, Data>::Node\* node)

{

if (!node) return NULL;

else if (node->left) // node = node->left;

{

node = node->left;

while (node->right)

node = node->right;

return node;

}

else

{

node = rightParent(r, node);

return node;

}

}

template <typename Key, typename Data>

typename Tree<Key, Data>::Node\* Tree<Key, Data>::searchHelper(typename Tree<Key, Data>::Node \*t, int k)

{

int m;

if (t==NULL) return t;

if (t->left==NULL) { m=0; }

else m=t->left->n;

if (m > k)

return searchHelper(t->left, k);

if (m < k)

return searchHelper(t->right, k-m-1);

return t;

}

template <typename Key, typename Data>

Key Tree<Key, Data>::search(int k)

{

Node \*t = searchHelper(root, k);

if(t==NULL) throw -1;

return t->key;

}

template<typename Key, typename Data>

typename Tree<Key, Data>::Iterator Tree<Key, Data>::begin()

{

if (!root)

throw exception("Исключение: дерево пустое");

Node\* temp = root;

while (temp->left)

temp = temp->left;

Iterator iter(this, temp);

return iter;

}

template<typename Key, typename Data>

typename Tree<Key, Data>::Iterator Tree<Key, Data>::rbegin()

{

if (!root)

throw exception("Исключение: дерево пустое");

Node\* temp = root;

while (temp->right)

temp = temp->right;

Iterator iter(this, temp);

return iter;

}

template<typename Key, typename Data>

typename Tree<Key, Data>::Iterator Tree<Key, Data>::end()

{

if (!root)

{

Iterator iter;

return iter;

}

Node\* temp = root;

while (temp)

temp = temp->right;

Iterator iter(this, temp);

return iter;

}

template<typename Key, typename Data>

Tree<Key, Data>::Iterator::Iterator()

{

tree = NULL;

node = NULL;

}

template<typename Key, typename Data>

Tree<Key, Data>::Iterator::Iterator(Tree<Key, Data>\* t, Node\* n)

{

tree = t;

node = n;

}

template<typename Key, typename Data>

Tree<Key, Data>::Iterator::Iterator(const Iterator& iter)

{

tree = iter.tree;

node = iter.node;

}

template<typename Key, typename Data>

Data& Tree<Key, Data>::Iterator::operator\*()

{

if (node)

return node->data;

else throw exception("Исключение: итератор за пределами дерева");

}

template<typename Key, typename Data>

typename Tree<Key, Data>::Iterator& Tree<Key, Data>::Iterator::operator++(int)

{

if (isInside())

node = tree->child(tree->root, node);

else throw exception("Исключение: итератор за пределами дерева");

return \*this;

}

template<typename Key, typename Data>

typename Tree<Key, Data>::Iterator& Tree<Key, Data>::Iterator::operator--(int)

{

if (isInside())

node = tree->parent(tree->root, node);

else throw exception("Исключение: итератор за пределами дерева");

return \*this;

}

template<typename Key, typename Data>

bool Tree<Key, Data>::Iterator::operator==(Iterator& iterator)

{

return node == iterator.node;

}

template<typename Key, typename Data>

bool Tree<Key, Data>::Iterator::operator!=(Iterator& iterator)

{

return node != iterator.node;

}

template<typename Key, typename Data>

bool Tree<Key, Data>::Iterator::isInside()

{

if (node == NULL) return false;

else return true;

}

template <typename Key, typename Data>

int Tree<Key, Data>::getCount()

{

int temp = cnt;

cnt = 0;

return temp;

}

**main.cpp**

#pragma once

#include <iostream>

#include "Tree.h"

#include "RBTree.h"

using namespace std;

namespace TestEffect

{

void test\_rand(int n);

void test\_sort(int n);

typedef unsigned long long INT\_64;

INT\_64 lrand() { return rand() << 16 | rand(); }

}

void main()

{

RBTree<int, int> tree;

RBTree<int, int>::Iterator it = RBTree<int, int>::Iterator(&tree, 0);

setlocale(LC\_ALL,"RUS");

while(true)

{

cout << endl << endl;

cout << "1. Просмотр дерева" << endl;

cout << "2. Очистка дерева" << endl;

cout << "3. Проверка дерева на пустоту" << endl;

cout << "4. Поиск узла с заданным ключом" << endl;

cout << "5. Добавление данных с заданным ключом" << endl;

cout << "6. Удаление данных с заданным ключом" << endl;

cout << "7. Опрос размера дерева" << endl;

cout << "8. Установка итератора на начало - begin()" << endl;

cout << "9. Установка итератора в конец - rbegin()" << endl;

cout << "10. Проверка состояния итератора" << endl;

cout << "11. Значение элемента на позиции итератора" << endl;

cout << "12. Запись значения на позицию итератора" << endl;

cout << "13. Iterator ++" << endl;

cout << "14. Iterator --" << endl;

cout << "15. Тест трудоемкости операций случайного дерева" << endl;

cout << "16. Тест трудоемкости операций вырожденного дерева" << endl << endl;

int t;

int value, key;

cout << "Ваш выбор: ";

cin >> t;

switch(t)

{

case 1:

cout << endl << endl;

tree.printTree();

break;

case 2:

tree.clear(tree.getRoot());

break;

case 3:

cout << tree.isEmpty() << endl;

break;

case 4:

cout << "Введите ключ: ";

cin >> key;

try

{

cout << tree.getData(key) << endl;

}

catch (...) { cout << "Исключение" << endl; }

break;

case 5:

cout << "Введите значение: ";

cin >> value;

cout << "Введите ключ: ";

cin >> key;

cout << tree.insert(value, key) << endl;

break;

case 6:

cout << "Введите ключ: ";

cin >> key;

cout << tree.deleteByKey(key) << endl;

break;

case 7:

cout << "Текущий размер дерева: " << tree.getSize() << endl;

break;

case 8:

try

{

it = tree.begin();

}

catch (exception& e) { cout << e.what(); }

break;

case 9:

try

{

it = tree.rbegin();

}

catch (exception & e) { cout << e.what(); }

break;

case 10:

cout << "Внутри дерева: " << it.isInside() << endl;

break;

case 11:

try

{

int\* data = &(it.operator\*());

cout << \*data << endl;

}

catch (exception & e)

{

cout << e.what() << endl;

}

break;

case 12:

cout << "Введите значение: " << endl;

cin >> value;

try

{

\*it = value;

}

catch (exception & e) { cout << e.what(); }

break;

case 13:

try

{

it++;

}

catch (exception & e) { cout << e.what(); }

break;

case 14: try

{

it--;

}

catch (exception & e) { cout << e.what(); }

break;

case 15:

int n;

cout << "Введите размер дерева:" << endl;

cin >> n;

TestEffect::test\_rand(n);

break;

case 16:

int n2;

cout << "Введите размер дерева:" << endl;

cin >> n2;

TestEffect::test\_sort(n2);

break;

default:

return;

}

}

}

void TestEffect::test\_rand(int n)

{

RBTree<int, INT\_64> tree;

Tree<int, INT\_64> BSTtree;

INT\_64\* m = new INT\_64[n];

for (int i = 0; i < n; i++)

{

m[i] = lrand();

tree.insert(1, m[i]);

BSTtree.setData(1, m[i]);

}

tree.getCount();

cout << "Размер дерева до: " << tree.getSize() << endl;

cout<<endl;

double I = 0;

double D = 0;

double S = 0;

double Ib=0;

double Db=0;

double Sb=0;

for (int i = 0; i < n / 2; i++)

if (i % 10 == 0)

{

tree.deleteByKey(lrand());

BSTtree.remove(lrand());

D += tree.getCount();

Db += BSTtree.getCount();

tree.insert(1, m[rand() % n]);

BSTtree.setData(1, m[rand() % n]);

I += tree.getCount();

Ib += BSTtree.getCount();

try

{

tree.getData(lrand());

BSTtree.getData(lrand());

S += tree.getCount();

Sb += BSTtree.getCount();

}

catch (...)

{

S += tree.getCount();

Sb += BSTtree.getCount();

}

}

else

{

int ind = rand() % n;

tree.deleteByKey(m[ind]);

BSTtree.remove(m[ind]);

D += tree.getCount();

Db += BSTtree.getCount();

INT\_64 key = lrand();

tree.insert(1, key);

BSTtree.setData(1, key);

I += tree.getCount();

Ib += BSTtree.getCount();

m[ind] = key;

try {

tree.getData(m[rand() % n]);

BSTtree.getData(m[rand() % n]);

S += tree.getCount();

Sb += BSTtree.getCount();

}

catch (...)

{

S += tree.getCount();

Sb += BSTtree.getCount();

}

}

cout << "Размер RB-дерева после: " << tree.getSize() << endl;

cout << "1.002\*log2(n)=" << 1.002\*(log((double)n) / log((double)2)) << endl;

cout << "Вставка: " << I / (n / 2) << endl;

cout << "Удаление: " << D / (n / 2) << endl;

cout << "Поиск: " << S / (n / 2) << endl;

cout<< endl;

cout << "Размер BST-дерева после: " << BSTtree.getSize() << endl;

cout << "1.39\*log2(n)=" << 1.39\*(log((double)n) / log((double)2)) << endl;

cout << "Вставка: " << Ib / (n / 2) << endl;

cout << "Удаление: " << Db / (n / 2) << endl;

cout << "Поиск: " << Sb / (n / 2) << endl;

delete[] m;

}

void TestEffect::test\_sort(int n)

{

RBTree<int, INT\_64> tree;

Tree<int, INT\_64> BSTtree;

INT\_64\* m = new INT\_64[n];

for (int i = 0; i < n; i++)

{

m[i] = i \* 1000;

tree.insert(1, m[i]);

BSTtree.setData(1, m[i]);

}

tree.getCount();

BSTtree.getCount();

cout << "Размер дерева до: " << tree.getSize() << endl;

cout<<endl;

double I = 0;

double D = 0;

double S = 0;

double Ib=0;

double Db=0;

double Sb=0;

for (int i = 0; i < n / 2; i++)

{

if (i % 10 == 0)

{

int k = lrand() % (1000 \* n);

k = k + !(k % 2);

tree.deleteByKey(k);

BSTtree.remove(k);

D += tree.getCount();

Db += BSTtree.getCount();

tree.insert(1, m[rand() % n]);

BSTtree.setData(1, m[rand() % n]);

I += tree.getCount();

Ib += BSTtree.getCount();

k = lrand() % (1000 \* n);

k = k + !(k % 2);

try {

tree.getData(k);

BSTtree.getData(k);

S += tree.getCount();

Sb += BSTtree.getCount();

}

catch (...)

{

S += tree.getCount();

Sb += BSTtree.getCount();

}

}

else

{

int ind = rand() % n;

try

{

tree.getData(m[rand() % n]);

BSTtree.getData(m[rand() % n]);

S += tree.getCount();

Sb += BSTtree.getCount();

}

catch (...)

{

S += tree.getCount();

Sb += BSTtree.getCount();

}

tree.deleteByKey(m[ind]);

BSTtree.remove(m[ind]);

D += tree.getCount();

Db += BSTtree.getCount();

int k = lrand() % (1000 \* n);

k = k + k % 2;

tree.insert(1, k);

BSTtree.setData(1, k);

I += tree.getCount();

Ib += BSTtree.getCount();

m[ind] = k;

}

}

cout << "Размер RB-дерева после: " << tree.getSize() << endl;

cout << "1.002\*log2(n)=" << 1.002\*(log((double)n) / log((double)2)) << endl;

cout << "Вставка: " << I / (n / 2) << endl;

cout << "Удаление: " << D / (n / 2) << endl;

cout << "Поиск: " << S / (n / 2) << endl;

cout<<endl;

cout << "Размер BST-дерева после: " << BSTtree.getSize() << endl;

cout << "n/2=" << n / 2 << endl;

cout << "Вставка: " << Ib / (n / 2) << endl;

cout << "Удаление: " << Db / (n / 2) << endl;

cout << "Поиск: " << Sb / (n / 2) << endl;

delete[] m;

}